Rapport de Projet de fin d'études

Projet BigBrowser

26/01/2011

Réalisé par Baptiste Carvaillo, Xavier Demompion et Féliks Plawczyk

Sommaire

Introduction : 4
Présentation de l'équipe et des différents acteurs du projet : 4
1) Présentation du sujet : 4
Contexte commercial : 4
Cas d'utilisation : 4
2) Présentation de la solution : 5
Exemple d'utilisation/résultat final : 5
Mode d'emploi : 5
3) Choix Technologiques : 6
Client : 6
La communication avec le serveur : 6
Serveur : 10
La communication avec le client ; 10
Partie Serveur - choix technologiques et fonctionnalités 13
Bibliographie : 24
Annexes : 25

Table des illustrations :

Figure 1 : Architecture Générale 6
Figure 2 : Menu d'accueil et fenêtre de dialog 8
Figure 3 : Architecture serveur actuelle 13
Figure 4 : Interface Ibrowser : 13
Figure 5 = Figure 3 : Architecture serveur actuelle 17
Figure 6 : Architecture serveur envisageable 18
Figure 7 : Architecture proposée dans le sujet/cahier des charges 21
Figure 8 : Synergie et pertinence 23

“By giving people the power to share, we're making the world more transparent.”

Mark Zuckerberg

Introduction :

Présentation de l'équipe et des différents acteurs du projet :

Le projet BigBrowser est réalisé par Xavier DEMOMPION, Féliks PLAWCZYK et Baptiste CARVAILLO. Il est encadré par Amel BOUZEGHOUB, enseignante-chercheur à Télécom-Sudparis et Djamel BONNEFFOUF thésard et employé de Nomalys, client du projet.

Ce projet à été réalisé dans le cadre du projet de fin d'étude de l'UV Architecte de Service en réseau sur une durée de 3 mois.

1) Présentation du sujet :

Contexte commercial :

Pour une entreprise connaissant la forte mobilité de ses commerciaux et leurs dépendances aux informations contenues dans la/les bases de données de ladite entreprise, il est très intéressant d'équiper l'ensemble de son personnel mobile d'une application permettant d'accéder la base de données d'entreprise à n'importe quel endroit. Dans ce cadre, il est pertinent d'envisager une application pour smartphone offrant cette possibilité.

Cas d'utilisation :

Le projet porte sur la réalisation d'un moteur de recherche contextuel d'entreprise, le but est de permettre aux utilisateurs de rechercher une entreprise ou un interlocuteur dans la base de données. La réponse est donnée sous forme d'une liste triée de fiches qu'ils peuvent consulter.
Cette application vise à offrir des résultats plus pertinents qu'une simple recherche dans une base de données. En intégrant la géolocalisation et un historique de toutes les recherches de tous les utilisateurs Bigbrowser offrira des résultats plus adaptés.
Cette application est destinée à des commerciaux en déplacement pour rendez vous. BigBrowser leur permettra de consulter les fiche contact de la base de donnée de leur entreprise et d'accéder plus vite à des informations pertinentes pour eux.

2) Présentation de la solution :

Exemple d'utilisation/résultat final :

L'utilisation de cette application est très simple, car elle est destinée à des commerciaux. Le but est donc de fournir un service utilisable de manière intuitive.
Notre production étant un « Proof of Concept », l'ergonomie
La première étape (après le lancement de l'application sur son smartphone Android bien entendu) est de s'authentifier.
L'utilisateur final doit donc s'authentifier en premier lieu, puis ensuite effectuer une ou des recherches. Une fois la recherche effectuée, l'application affiche le résultat par ordre de pertinence.
Il a également certaines possibilités concernant la configuration de l'application.

Mode d'emploi :

- Lancer l'application
-S'authentifier avec son login et son mot de passe
- Effectuer la recherche
- Apprécier les résultats

Architecture : Vue Globale.

Nous avons opté pour une architecture à trois tiers, dont voici une description globale. Les choix technologiques seront détaillés dans la partie suivante.

Le premier tier est destiné à l'utilisateur. Il s'agit du client développé sur Android. C'est l'interface entre l'utilisateur et le service, il n'y a aucun traitement de données ici. Le client sert à récupérer le contexte physique de l'utilisateur (identité et géolocalisation) ainsi que sa requête.

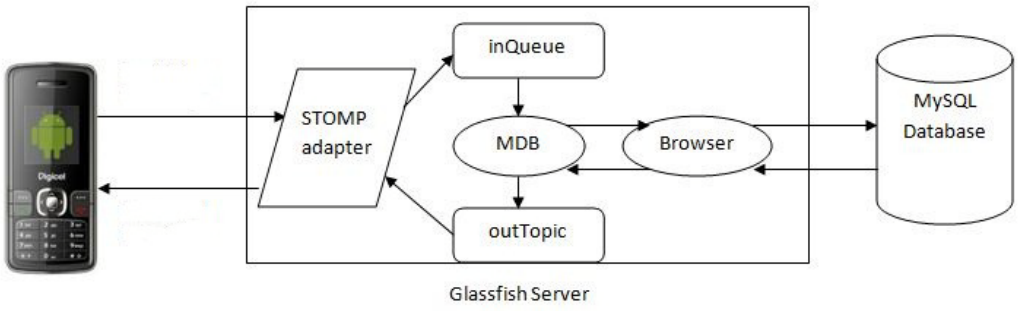
Le second est le serveur. Il s'occupe de traiter les requêtes du client en effectuant une recherche prenant en compte le contexte envoyé avec la requête. Ce serveur communique à la fois avec le client et les bases de donnée.

Le troisième tier est donc composé d'une (deux) base de donnée, il s'occupe de stocker toutes les informations de contact, ainsi que celles utilisées par BigBrowser pour ses recherches.

L'intérêt d'une telle architecture est la facilité de développement en équipe, chaque tier offrant une interface aux autres. Cela permet aussi de modifier un unique tier sans avoir besoin de retoucher à toute l'application.

3) Choix Technologiques :

Figure 1 : Architecture Générale



Client :

Le programme client est développé sous Android 2.2

La communication avec le serveur :

Il communique avec le serveur grâce au protocole STOMP (Simple Text Oriented Message Protocol).
La librairie Gozira fournit une implémentation simple de STOMP, permettant d'envoyer des trames STOMP à un serveur.

Une trame STOMP est constituée de 3 éléments :
- Une commande
- des Headers
- un corps composé de texte

Les commandes qui nous intéressent sont SEND pour envoyer des messages, SUBSCRIBE pour s'abonner à une destination et écouter dessus et UNSUBSCRIBE pour se désabonner.
Le client permet de s'affranchir de l'écriture de la totalité de la trame et s'occupe de gérer les sockets pour la connexion au point distant.
Afin de faciliter le traitement des messages, nous avons utilisé la librairie JSON pour pouvoir aisément transmettre des paquets STOMP du client au serveur via TCP. Ainsi chaque requête est en fait un objet transformé en String à l'aide de JSON puis envoyé au serveur dans un paquet STOMP. Bien évidemment, chaque message provenant du serveur utilise la même solution.

Le client émet deux types de requêtes, les LoginReq, afin de s'authentifier et les DataReq, pour effectuer les recherches sur la base de données. Ces deux types de requêtes sont en

fait des objets envoyés au serveur dans une trame STOMP, en étant convertit en String au préalable à l'aide de la librairie JSON. A chaque requête, le client s'abonne au topic de réponse du serveur et se met en attente d'une réponse lui étant destinée afin de la traiter. Dans le cas d'une LoginReq, il reçoit une LoginReq traitée, avec un booléen représentant le succès ou non de l'authentification. Dans le cas d'une DataReq, le client attend un objet ResultSet dans lequel il y a les informations demandées et triées par pertinence.

Toutes ces communications se font par le biais du même objet, le StompSender. Il offre une interface simple comprenant deux méthodes, login et search, et revoyant les résultats sous forme de booléen ou Collection de données respectivement. Ainsi on peut dissocier le développement de l'interface utilisateur de la communication avec le serveur. La seule charnière entre les deux étant une interface simple.

Le StompSender :

C'est ce qui permet au client de se connecter au serveur, d'envoyer et de recevoir des données.

Lors de la création de l'objet StompSender, on instancie un client STOMP qui se connecte à notre serveur et attend des appels de méthode d'envoi.
Ce StompSender propose deux méthodes pour les échanges :
- login
- search

Leurs noms sont explicites, on utilise login pour s'authentifier et search pour effectuer une recherche.

Sur un appel de la méthode login, le StompSender s'abonne au topic de réponse du serveur, puis crée un objet LoginReq et l'envoie. Il se met ensuite en attende d'une réponse sur le topic, dès qu'il reçoit une trame STOMP, si c'est une LoginReq, il la décode et vérifie qu'elle lui est adressée, puis si la réponse lui est adressée, il récupère le résultat de la requête et renvoie un booléen (true = succès du login bien évidemment).

Sur un appel de la méthode search c'est le même principe, sauf qu'il envoie un objet DataReq et se met en attente d'un ResultSet. Il renvoie une Collection ordonnée de résultats que le client pourra afficher.

Le traitement des données et l'interface sous Android

Les fonctionnalités implémentées et le code actuel :

architecture générale :

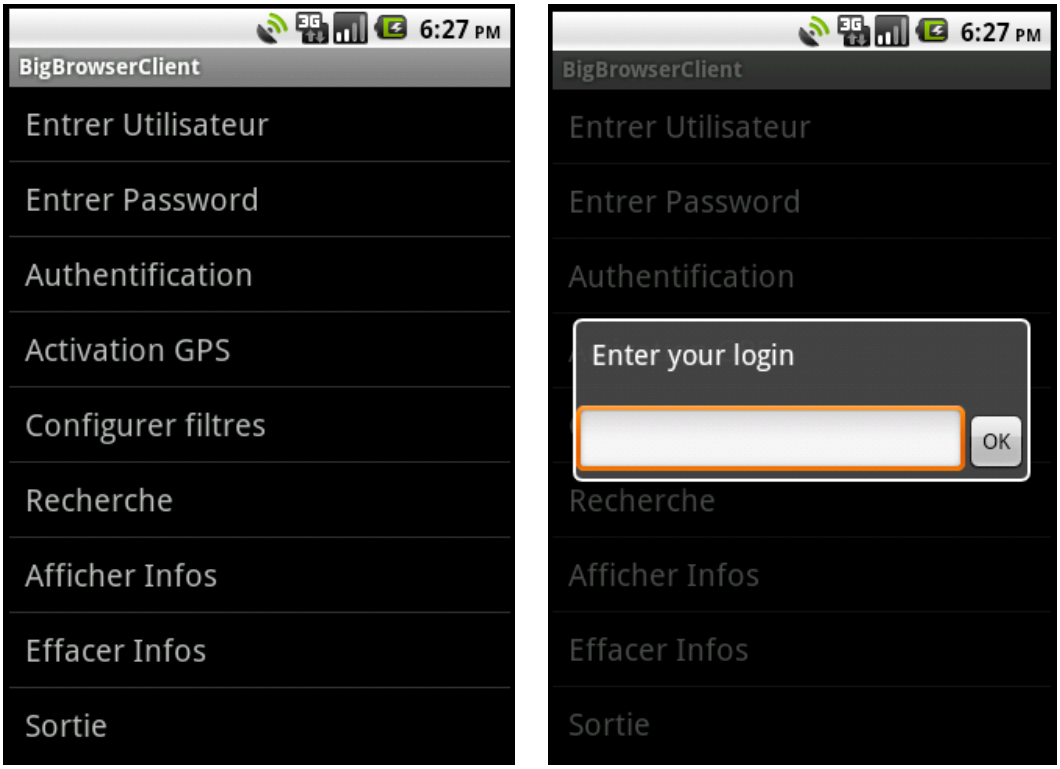
Le client utilise le StompSender présenté précédemment pour se connecter au serveur et obtenir des informations (autorisation d'accès, résultat d'une requête)

L'interface est donc composée d'un premier menu proposant plusieurs possibilités, en particulier, toutes les fonctions de configuration et la possibilité de sortir du système ou de lancer une recherche. Toutes les interactions entre l'utilisateur et le système se font par l'intermédiaire de fenêtres de dialogue, et des fenêtres "Toast" (équivalent des pop-ups) apportent des informations de manière extrêmement simple à l'utilisateur lors de certaine actions. Elles sont en particulier utilisées pour informer l'utilisateur lors d'une erreur en identifiant ladite erreur.

Enfin, lors du lancement de la recherche, on accède à notre deuxième interface qui permet d'afficher de manière synthétique la liste de résultats. Cette deuxième interface est démarrée dans une activity différente de la principale. On lui passe donc par l'intermédiaire d'un Bundle toutes les données dont elle à besoin.

Ces deux listes (le menu ou l'affichage du résultat) ont cela en commun qu'elles sont définies de manières dynamique. Cela n'est pas forcément obligatoire pour le menu mais permet une plus grande souplesse lors de l'écriture du code, le menu pouvant être modifié par une simple modification d'un tableau de String.

Figure 2 : Menu d'accueil et fenêtre de dialog



les fonctions en détail :

La fonction authentification (login, mot de passe et authentification) :

Le choix est ici d'utiliser des fenêtres de Dialog ,fenêtres qui ont l'avantage d'être implémentées simplement et s'affichent en surcouche de la .

Les avantages lors du développement sont multiples : possibilité d'utiliser de manière simple une même implémentation de listener pour modifier des données, possibilités de réutilisation du code pour plusieurs fonctionnalités.

En effet, il y a ici possibilité de n'utiliser qu'une seule interface (un seul layout), et d'écrire plusieurs classes de dialog différentes et personnalisées faisant appel à ce layout. Ainsi, les deux fenêtres qui invitent à entrer les login et mot de passe de l'utilisateur font appel à la même classe et utilisent le même layout. Elles sont simplement crées avec des paramètres différents.

Ensuite, lors du clic sur l'élément authentification, on appelle la méthode stompsender.login qui s'occupe d'interroger le serveur et de retourne le résultat qui nous intéresse.

La fonction recherche :

La fonction recherche effectue plusieurs actions importantes de manière séquentielles. Elle va tout d'abord afficher une fenêtre de dialog afin de pouvoir récupérer la requête de l'utilisateur (un objet de type string).
Le listener liée au bouton sur cette fenêtre de dialog va ensuite initier la communication avec le serveur en faisant appel à la fonction stompsender.search(arguments qu'il faut) et recevoir une liste de link., objet contenant l'ensemble des informations liées à un contact, entreprise et ou personne.
Cette liste de link est ensuite traitée afin d'être passée en argument de l'activity qui affiche notre liste de résultats et permet d'interagir avec ceux-ci.

les fonction Activation GPS (et donc getGPS) :

Au clic sur cette partie du menu, on va lancer une méthode qui va essayer d'activer le GPS et d'implémenter un listener sur le provider. Cependant, comme nous savons que le GPS est extrêmement consommateur d'énergie et potentiellement peut consommer énormément de batterie et donc que l'utilisateur puisse le désactiver, ou que l'utilisateur possède un téléphone ne faisant pas GPS, notre fonction getGPS va lister l'ensemble des éléments qui peuvent fournir à l'application la position même approximative du téléphone.

Ensuite, on va initialiser une première fois les coordonnées GPS en parcourant la liste et en prenant la première valeur existante de la dernière position connue, tout en donnant priorité au GPS s'il est disponible.
Par la suite, si le GPS est disponible sur le téléphone, on va créer un listener et vérifier périodiquement la position du téléphone.

la fonction affichage information est uniquement là pour pouvoir afficher l'état des variables. C'est en quelque sorte un "ToString" pour le développement et les démos.

La fonction Sortie est une méthode qui fait appel à la fonction onfinish(); Elle ne sert à rien si ce n'est à remplacer si on le désire le bouton retour de l'android.

Possibilités d'évolutions de l'application :

Optimisation du code et amélioration de l'implémentation de certaines fonctions :

Plusieurs fonctions mériteraient d'être retouchées, soit par souci d'ergonomie, soit afin de permettre au programme de répondre plus précisément aux demandes de l'utilisateur, soit enfin, parce qu'il doit probablement exister des bugues que nous n'avons pas identifié.

La partie concernant l'authentification

Cette partie souffre d'un problème d'ergonomie, en effet, il faut d'abord rentrer le login, puis le password dans deux fenêtres séparées avant de pouvoir lancer l'authentification. Cela serait probablement plus pertinent de tout regrouper dans une unique interface. Cependant, comme beaucoup de remarque qui seront dans cette partie, cela n'a pas été fait car n'apporte rien d'intéressant dans la partie fonctionnalité du code. C'est une considération uniquement ergonomique.

Fonctionnalités non développées fautes de temps et nouvelles fonctionnalités à envisager.

-config des filtres.
-possibilité de call direct
-affichage complet des liens
-affichage du plan pour se rendre à un des lien en utilisant l'api google map :

Il serait en effet intéressant de pouvoir à la demande, lors de la sélection d'un lien de proposer le plan d'accès en utilisant l'api google map.
Cela serait en effet intéressant de pouvoir profiter pleinement de l'intérêt d'être sur android. Cependant, il faudrait lors de l'implémentation de ce type de service être vigilant à la manière dont elle est implémentée afin de respecter les restrictions (principalement sur le nombre de requêtes par jour) liées à l'utilisation des services web de google.

Serveur :

La communication avec le client ;

Nous avons opté pour un serveur Glassfish, nous permettant de développer en J2EE et possédant quelques avantages que nous allons détailler maintenant.

Pourquoi Glassfish?

Ayant plus de connaissances en développement en J2EE qu'en web service, nous avons choisit la relative facilité des Entreprise Java Beans. Glassfish étant le serveur sur lequel nous avions déjà travaillé, il nous a semblé judicieux de le choisir pour ce projet. Voici donc les multiples raison raisons qui ne nous ont pas fait regretter de choisir Glassfish v3 :
Tout d'abord c'est un conteneur de JavaBeans, il en faut obligatoirement un pour déployer une application en J2EE.

Il possède un adaptateur STOMP <-> JMS, nous permettant de traiter facilement les requêtes de un ou plusieurs clients sous Android.

Il est possible de laisser Glassfish gérer la persistance des données, notamment toutes celles relatives à l'histoire des requêtes des utilisateurs via les Entity Beans.

Pourquoi STOMP?

Android n'implémente que peu de bibliothèques java à cause de la spécificité de sa machine virtuelle. Ayant choisit d'utiliser des EJB du coté du serveur, il nous fallait un moyen simple pour pouvoir envoyer des messages depuis un client sur Android à un serveur. La solution la plus facile d'accès est d'utiliser STOMP. En effet, Glassfish possède nativement un adaptateur STOMP-JMS et la bibliothèque Gozira, fournissant un client STOMP est compatible avec Android.

L'application serveur est composée de 2 Beans, une queue JMS d'entrée de messages et un topic JMS pour les réponses.

Le premier bean est un message driven bean, il écoute sur la queue d'entrée et traite tout les messages. Il fait appel au second bean pour appliquer les algorithmes de recherche et interroger la base de données. Une fois le traitement effectué, il s'occupe de transmettre la réponse au client via le topic auquel les clients sont abonnés.

Architecture JMS :

Le serveur écoute sur une Queue nommée jms.inQueue. Tout les messages de tout les clients arrivent dans cette queue et sont traités par un unique message driven bean. Afin de traiter chaque requête ce bean fait appel à un stateless bean qui gère les connections à la base de donnée et le traitement de la requête.

Chaque message reçu est un message texte contenant l'objet de requête formaté en JSON. La première étape du traitement est de détecter s'il s'agit d'une requête de login ou de données. Le bean regarde donc dans le champ "classe" de l'objet JSON reçut afin de pouvoir déterminer quel est le type de requête. Après cette étape, le message driven bean reconstruit l'intégralité de l'objet reçut et peut commencer le traitement. Ensuite ce bean fait appel au stateless bean définit par l'interface IBrowser, s'occupant du traitement. Une fois ce traitement effectué, le message driven bean construit l'objet réponse (un LoginReq ou un ResultSet) et l'envoie aux clients abonnés au topic de réponse.

Faute de temps pour implémenter des topics temporaires pour chaque utilisateur, le serveur envoie toutes les réponses sur le même topic et c'est le client qui ignore les messages qui ne lui sont pas destinés. L'architecture est donc simple, mais peu scalable en nombre d'utilisateur à cause de la quantité de messages inutiles.

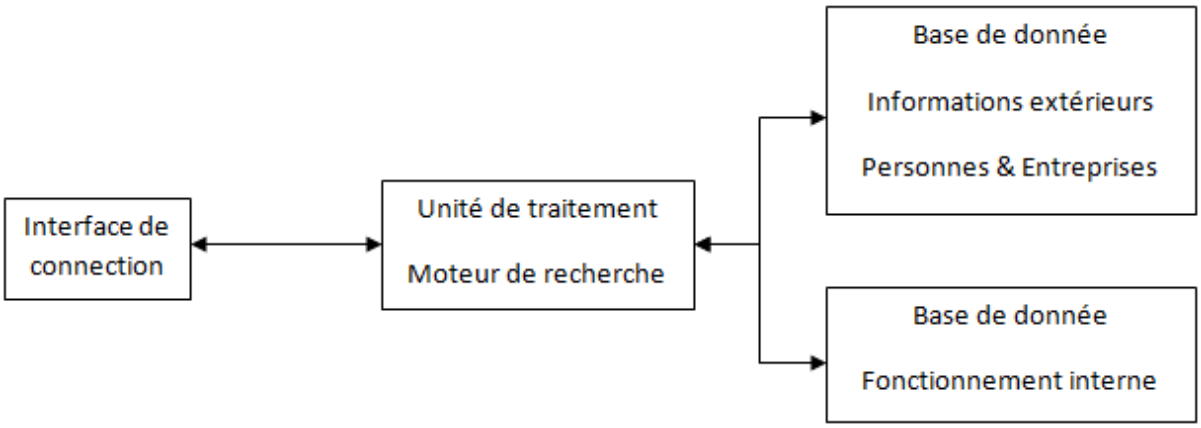
Pourquoi 2 beans différents?

Séparer le traitement en deux beans différents permet de développer presque indépendamment l'aspect traitement de données et communication. Le seul dénomminateur commun est la classe décrivant les données qui doit comporter les méthodes nécessaires à la sérialisation à l'aide de JSON. Pour le reste, tout ce que voit le message driven bean est une interface qui retourne les données qui l'intéressent (orthographe???). Dans le cas d'un login, le service retourne un booléen (true = succès bien sûr), dans le cas d'une recherche il retourne une collection triée de Link, chaque Link représentant des données de la base de donnée. Cela permet de changer les méchaniques de traitement sans pour autant changer la gestion des messages, ni même le client. Seules des modifications de la classe Link doivent être répercutées sur les 3 tiers.

Partie Serveur - choix technologiques et fonctionnalités

Description du service

Figure 3 : Architecture serveur actuelle



Interface du service

C'est la porte d'entrée de notre service, elle permet à n'importe quel utilisateur de faire appel à chacune des fonctionnalités du service.

Figure 4 : Interface Ibrowser :

```
@Remote
public interface IBrowser {
    //log in
    public boolean isUserPswdExist(String login,String password);
    // search engine
    public Collection<Link> search(String requestString, String login, userContext myContext,
    double distanceW,double socialW, double profilW, double historiqueW);
    // log - data feeder
    public void publishActionOnLink(String originalRequest, userContext myContext, String login,
            String action, Link link);
}
```

Le nombre de méthodes est très réduit :

isUserPswdExist() permet de s'authentifier.

search() est la fonctionnalité de recherche, elle retourne une collection de Link (lien).

publishActionOnLink() est une méthodes appeler à chaque action du client. Cela permet d'enrichir l'historique, les éléments les plus visités.

Descriptif détaillé :

- public boolean isUserPswdExist(String login,String password);

L'utilisateur fourni son login et mot de passe par cette fonction, si il est bien présent, il reçoit un "true" en retour.

- public Collection<Link> search(String requestString, String login, userContext myContext, double distanceW,double socialW, double profilW, double historiqueW);

L'utilisateur nous fourni :

- sa requête tapée dans le moteur de recherche (requestString)
- son login (login)
- son contexte, c'est à dire sa position, son modèle de téléphone, son heure locale .... (myContext)
- 4 poids qui permettent de demander au moteur de recherche de prendre plus où moins en compte un des quatre aspects suivant : distance géographique, réseau social, profil utilisateur, historique utilisateur

S'en retourne une collection de Link, un Link dispose de :

- un type (ex : entreprise ou personne)
- Informations relatives à l'entreprise ou à la personne
- un index : sa position dans la collection (plus l'index est petit, plus le lien en question est pertinent pour la recherche effectuée)
- un score : indice de pertinence de lien en question
- un indicateur qui nous dit quel élément à le plus apportée au score. Cet élément est soit le social, soit l'historique, soit le profil utilisateur, soit la proximité.

Ressources externes : bases de données

Nous utilisons MySql pour gérer nos bases de données.

- Les informations brutes

Nous disposons d'une base de données qui nous amène toutes informations relatives aux entreprises et aux personnes qui nous pourrions rechercher.

Parmi ces informations nous utilisons particulièrement :

- Nom de l'entreprise
- Adresse de l'entreprise
- Téléphone de l'entreprise
- Position GPS de l'entreprise
- Marché de l'entreprise (ex : télécom, alimentaire)
- type de l'entreprise (point de vente, site de production ...)

- Nom, Prénom de la personne
- Téléphone de la personne
- Position (ex : commercial, informaticien ...)

- Les informations utiles au fonctionnement

Nous utilisons les EJB (Entreprise Java Bean) pour gérer ces informations. Une table Sql correspond à un type d'objet (un bean).

Nous avons les bean suivants :

- KeyWord : mot clé définissant un profil utilisateur Log : action produite par un utilisateur enregistrée coté serveur pour être ré-exploitée
- shortCutSearch : système de raccourci de recherche défini par utilisateur
- User : informations relatives à l'utilisateur

L'utilisateur des EJB nous permet de manipuler très facilement toutes ces données, sans manipuler le sql. Nous manipulons des séries d'objets liés les uns aux autres.

Moteur de recherche

Notre moteur de recherche nous permet de chercher dans nos entreprises et nos personnes.

Ce que nous avons implémenté lorsqu'une recherche est effectuée :

- remplacement des mots clés par le mot clé associé grâce aux shortcuts le cas échéant.
- identification des mots clé : identification entreprises, personnes ...

- filtre brut qui nous donne uniquement les personnes et les entreprises correspondant aux mots clés
- On constitue la collection de Link
- Ordonnancement de nos résultats suivant les quatre critères de notre système : distance géographique, réseau social, historique utilisateur et profil utilisateur. Le tout grâce à un système de scoring.
- On renvoie la collection de Link ainsi rangée.

Système des quatre poids

De par ces quatre poids, nous mettons un ordre de pertinence de nos liens.

Nous allons ici les décrire un peu plus en détail :

Poids géographique :

Plus notre utilisateur est proche géographiquement du contact, plus le score est élevé.

Poids profil utilisateur :

Chaque utilisateur possède des mots clés qui le décrit. Nous comptons combien de mots clés sont en commun avec ce qui décrit les entreprises et les personnes à propos de leur marché, de leur position ou de leur type.

Poids réseau social :

Grâce au champ "like" de chaque entité, nous savons quelle entreprise ou quelle personne est populaire suite aux recherches faites par tous les utilisateurs. Nous avons donc un indice de popularité qui reflète par le réseau social l'importance de cette personne ou de cette entreprise.

Poids historique :

Grâce au Log, nous savons ce qu'a sélectionné notre utilisateur lors de ces recherches précédentes. Nous en sortons un score.

Chacun de ces quatre poids est calculé, puis pondéré en fonction des arguments données par l'utilisateur.

Un score final par Link est donné, puis nous trions en fonction de ce chiffre.

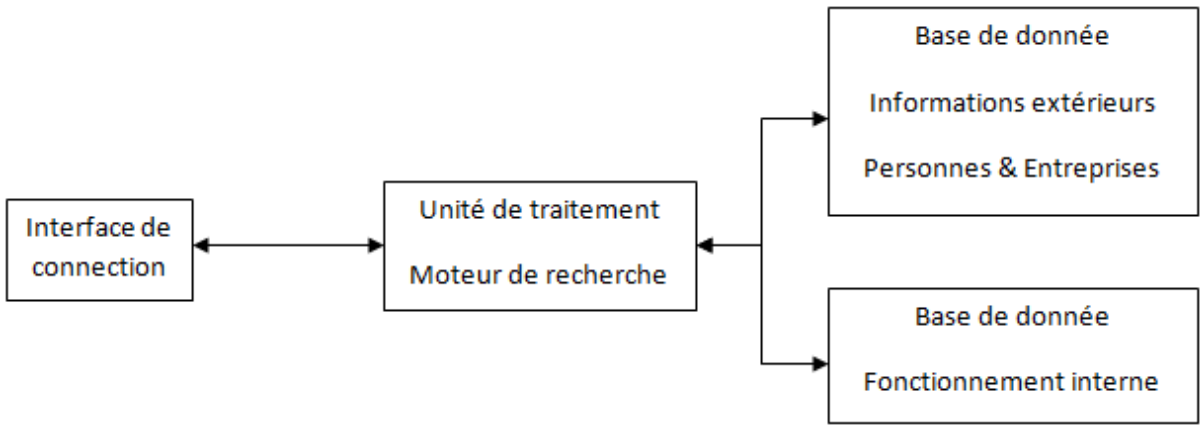
Améliorations futures :

scalabilité :S

Le service étant développé dans le monde java, nous avons de grandes possibilités d'évolution d'architectures s'il est question de scalabilitation.

L'architecture logique des acteurs du produit actuel est faite ainsi :

Figure 5 = Figure 3 : Architecture serveur actuelle



Impératif à respecter par bloc :

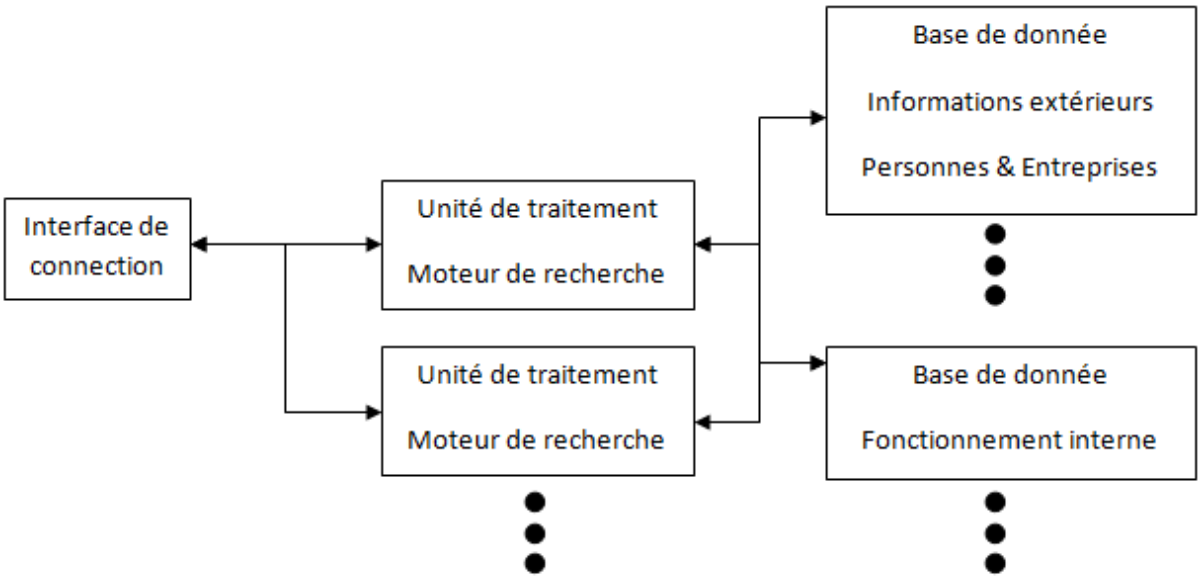
L'interface est unique

L'unité de traitement est totalement distribuable, elle n'a aucune besoin de synchronisation.

Les bases de données peuvent être distribuées, toutefois elles requièrent des synchronisations.

On pourrait donc tendre vers ce genre d'architecture :

Figure 6 : Architecture serveur envisageable



Optimisation :

Si le projet tente à vouloir se tourner vers une version industrialisable, il va falloir faire un effort de reconstruction sur certaine sous partie pour des raisons de performances.

En effet ce projet est orienté "proof of concept" c'est à dire que c'est un démonstrateur d'une solution novatrice et puissance, facilement modifiable suivant le produit final que nous voudrions avoir lors d'une hypothétique commercialisation.

Ontologie :

A moindre coût, notre produit peut facilement se greffer sur un système d'ontologie déjà existant.

Si nous voulons développer nous même notre ontologie, mais cela à un coût en homme heure.

Analyse critique du sujet initial

Dans cette partie nous allons analyser le produit ainsi créé lors de ce projet de fin d'étude.

Nous allons justifier pourquoi nous nous sommes détachés de certains aspects de la solution proposée par le sujet, ainsi que ce que nous avons proposé pour répondre plus correctement à la finalité du produit.

Base de donnée initiale - mise en place des ressources

La base de donnée échantillon proposée pour notre projet par l'industrielle était composée d'une dizaine de tables, dont deux nous étaient principalement utiles :

- une table société

- une table personne

La table société possède toutes les informations sur chaque société enregistrée. Notamment son adresse qui permet de retrouver la position GPS.

La table personne possède toutes les informations sur des personnes et des contacts potentiels qui peuvent être intéressant à contacter.

Ce format est satisfaisant, mais cette base de données échantillon se prête difficilement à la pertinence de notre produit :

BigBrowser fait des croisements d'informations entre différentes personnes dans différentes entreprises puis nous fourni parmi les personnes/entreprises trouvée celle les plus pertinentes selon la distance, le profil utilisateur, son historique et son réseau social.

Or la base de donnée échantillon en composée que d'entreprises isolées et des personnes qui n'ont pas vraiment de liens avec elles.

Ces tables étaient donc quasi non réutilisables.

Donc afin de mieux constaté l'efficacité de BigBrowser, nous avons produit une source d'information factice qui se prête à l'usage de notre application.

Nous avons donc créé un milieu à l'échelle d'une ville.

Nous avons créé 3 sociétés ayant chacune 3 bâtiments répartis dans cette ville.

Ensuite nous avons créé un personnel varié mais avec des points communs répartis dans ces 9 lieux de travail.

Solution initiale pour le trainement du moteur de recherche

Figure 7 : Architecture proposée dans le sujet/cahier des charges

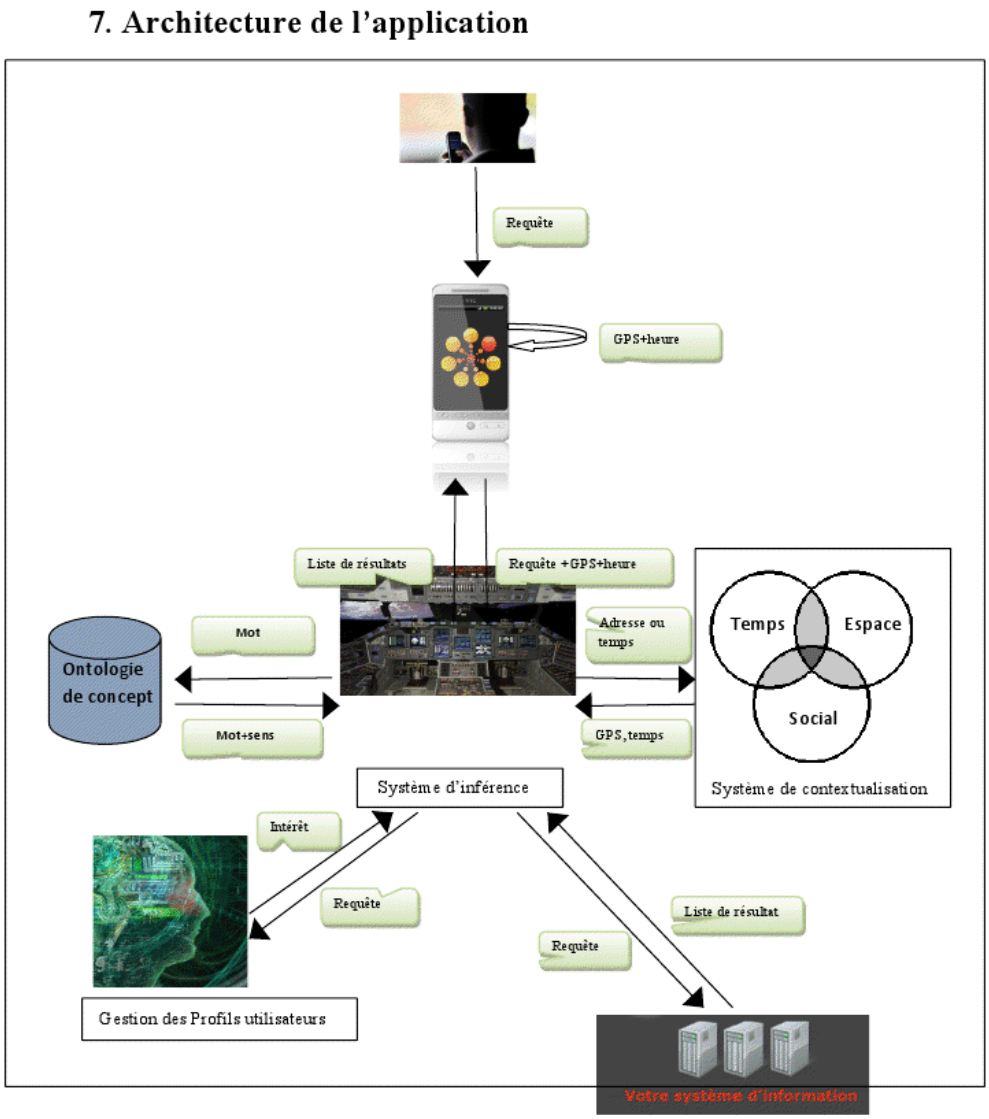


Figure 2 Architecture du moteur de recherche

5

Le sujet nous suggère d'utiliser les ontologies pour les processus suivants :

- reconnaissance des mots clés pour déduire si cela est une entreprise ou/et un personne.

- synergie de la recherche avec les données disponibles.

- synergies avec le profil utilisateur et son contexte.

Rapidement nous nous sommes aperçus que cela revenait à tuer un moustique avec une bombe atomique !

Pour la reconnaissance de mot clé, une simple requête sql suffit, nous faisons une recherche exclusive sur les mots clés, et nous savons très facilement par sql identifier chacun des mots clés.

Pour la synergie avec les données disponibles, cela sort du cadre des ontologies : nous voulons juste ordonnancer nos résultats suite au filtrage fait grâce aux mots clés.

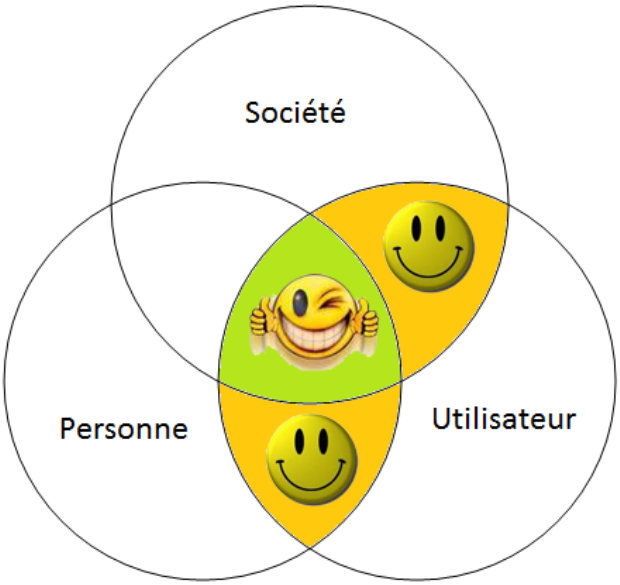
Nous utilisons donc uniquement des calculs de distances, de évaluations de profil, de l'historique, et du réseau social le tout grâce à des pondérations numériques modifiables par l'utilisateur, cela simplifie grandement les choses, en plus d'avoir une recherche plus pertinente pour notre type d'application.

Pour la synergie avec le profil utilisateur et son contexte, les ontologies fonctionnerait très bien (dans un cadre idéal) si nous prenons le temps de l'aboutir jusqu'au bout. Toutefois il faudrait interfacer les ontologies avec d'autre environnement très différent. Notre système serait donc très hétérogène et rendrait nos résultats avec une pertinence hasardeuse.

Point de fonctionnement nombre :

Notre solution profiterait d'un passage à l'échelle, en effet avec le nombre et la diversité des informations disponibles ainsi que les utilisateurs, nous sommes plus à même de fournir un nombre et une précision d'information grandissante.

Figure 8 : Synergie et pertinence



Bibliographie :

http://stomp.codehaus.org/ : Site sur le protocole STOMP
http://glassfish.java.net/ : Site du projet Glassfish
http://download.oracle.com/javaee/6/tutorial/doc/ : Tutoriel Java EE 6 (seuls les chapitres 14-17 et 27-31 ont été utiles)
http://developer.android.com/index.html : Site officiel de Android
http://www.xda-developers.com/ : Forum de développeur Android
http://www.instablogsimages.com/images/2009/06/19/solar-android-phone-zte_Gekxs_69.jpg
Source de l'image android des slides : http://www.touchmobile.fr/

Annexes :

Plan de charge :

tâches		Caravaillo Baptiste	Demompion Xavier	Plawczyk Féliks		tps total groupe
encadrement du projet		40	40	40		120
cahier des charges		5	5	5		15
choix des techno		10	10	10		30
configuration logicielle		10	10	10		30
réunions		15	15	15		45
Production		120	120	120		360
bases de données		10	20	0		30
modélisation du contexte		10	10	10		30
Mise en place de l'outillage logiciel						
	API' fonctions et toolbox	40	40	40		120
	interface utilisateur	10	0	20		30
	tests unitaires & validation	20	20	20		60
utilisation des ressources constituées	(application d'un usecase simple)	30	30	30		90
Préocupations scolaires		20	20	20		60
point synchrone avec la scolarité		10	10	10		30
synthèse, préparation orale & rapport		10	10	10		30
Total		180	180	180		540